# Threads / Synchronization 1

# Changelog

Changes made in this version not seen in first lecture:

22 September 2019: thread resources exercise (what's wrong with this): be more consistent about thread function name

30 September 2019: passing data structures: include mandatory return NULL on thread function

# last time

fair schedulers and proportional share

- lottery — random choice
- CFS — equalize ‘virtual times’

tradeoff: credit for time while not runnable

real-time schedulers

# lottery scheduler assignment

track "ticks" process runs

- = number of times scheduled
- simplification: don't care if process uses less than timeslice

new system call: getprocesesinfo

- copy info from process table into user space

new system call: settickets

- set number of tickets for current process
- should be inherited by fork

scheduler: choose pseudorandom weighted by tickets

- caution! no floating point

# thread versus process state

thread state — kept in **thread control block**

- registers (including program counter)
- other information?

process state — kept in **process control block**

- address space (memory layout)
- open files
- process id
- ...

# Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared — if same process

`fork()`-like system call "clone": choose what to share

`clone(CLONE_FILES, ...)` — new process **sharing** open files

`clone(CLONE_VM, ...)` — new process **sharing** address spaces

# Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared — if same process

`fork()`-like system call "clone": choose what to share

- `clone(CLONE_FILES, ...)` — new process **sharing** open files
- `clone(CLONE_VM, ...)` — new process **sharing** address spaces

advantage: no special logic for threads (mostly)

# aside: alternate threading models

we'll talk about kernel threads

OS scheduler deals directly with threads

alternate idea: library code handles threading

kernel doesn't know about threads w/in process

hierarchy of schedulers: one for processes, one within each process

not currently common model — awkward with multicore

# why threads?

concurrency: different things happening at once

- one thread per user of web server?
- one thread per page in web browser?
- one thread to play audio, one to read keyboard, ...?
- ...

parallelism: do same thing with more resources

- multiple processors to speed-up simulation (life assignment)

# pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

run ComputePi and PrintClassList at the same time

also run “more code”

# pthread_create

```
void
void
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

thread identifier — used to perform operations on thread later

run ComputePi and PrintClassList at the same time

also run “more code”

# pthread_create

function to run — thread starts here, terminate if function returns

```
vo
void *PrintClassList(void *argument) { ... }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

run ComputePi and PrintClassList at the same time

also run “more code”

# pthread_create

```
void *C
void *P
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

thread attributes (extra settings) and function argument

run ComputePi and PrintClassList at the same time

also run “more code”

# a threading race

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n");
    return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    return 0;
}
```

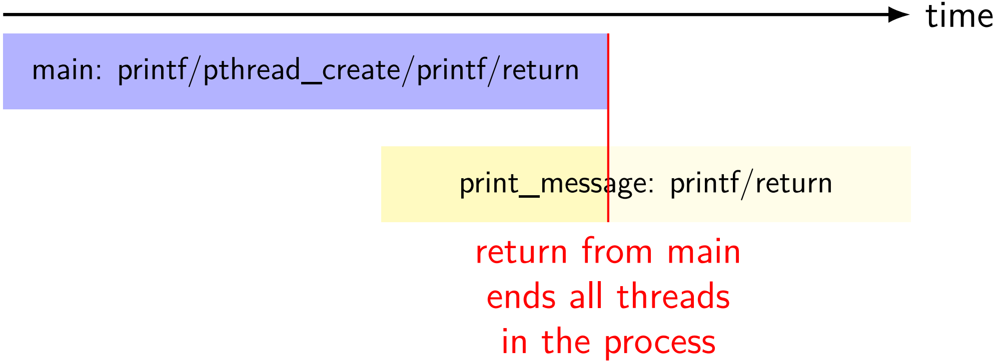
My machine: outputs `In the thread` about 4% of the time.

What happened?

# a race

returning from main exits the entire process (all threads)

race: main's return 0 or print_message's printf first?



# fixing the race (version 1)

```c
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n");
    return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    pthread_join(the_thread, NULL);  /* WAIT FOR THREAD */
    return 0;
}
```

# fixing the race (version 2; not recommended)

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n");
    return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    pthread_exit(NULL);
}
```

# pthread_join, pthread_exit

`pthread_join`: wait for thread, returns its return value

- like `waitpid`, but for a thread
- return value is pointer to anything

`pthread_exit`: exit current thread, returning a value

- like `exit` or returning from main, but for a single thread
- same effect as returning from function passed to pthread_create

# passing thread IDs (1)

```
DataType items[1000];
void *thread_function(void *argument) {
    int thread_id = (int) argument;
    int start = 500 * thread_id;
    int end = start + 500;
    for (int i = start; i < end; ++i) {
        DoSomethingWith(items[i]);
    }
    ...
}
void run_threads() {
    vector<pthread_t> threads(2);
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void*) i);
    }
}
```

# passing thread IDs (1)

```
DataType items[1000];
void *thread_function(void *argument) {
    int thread_id = (int) argument;
    int start = 500 * thread_id;
    int end = start + 500;
    for (int i = start; i < end; ++i) {
        DoSomethingWith(items[i]);
    }
    ...
}
void run_threads() {
    vector<pthread_t> threads(2);
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void*) i);
    }
}
```

# passing thread IDs (2)

```
DataType items[1000];
int num_threads;
void *thread_function(void *argument) {
    int thread_id = (int) argument;
    int start = thread_id * (1000 / num_threads);
    int end = start + (1000 / num_threads);
    if (thread_id == num_threads - 1) end = 1000;
    for (int i = start; i < end; ++i) {
        DoSomethingWith(items[i]);
    }
    ...
}
void run_threads() {
    vector<pthread_t> threads(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void*) i);
    }
    ...
}
```

# passing thread IDs (2)

```
DataType items[1000];
int num_threads;
void *thread_function(void *argument) {
    int thread_id = (int) argument;
    int start = thread_id * (1000 / num_threads);
    int end = start + (1000 / num_threads);
    if (thread_id == num_threads − 1) end = 1000;
    for (int i = start; i < end; ++i) {
        DoSomethingWith(items[i]);
    }
    ...
}
void run_threads() {
    vector<pthread_t> threads(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void*) i);
    }
    ...
}
```

# passing data structures

```
class ThreadInfo {
public:
    ...
};

void *thread_function(void *argument) {
    ThreadInfo *info = (ThreadInfo *) argument;
    ...
    delete info;
    return NULL;
}

void run_threads(int N) {
    vector<pthread_t> threads(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void *) new ThreadInfo(...));
    }
    ...
}
```

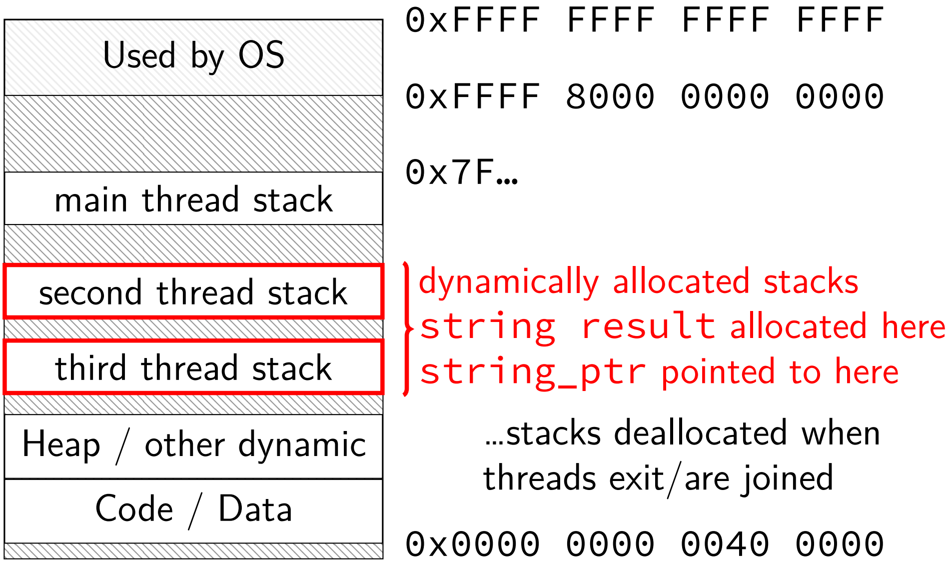
# passing data structures

```
class ThreadInfo {
public:
    ...
};

void *thread_function(void *argument) {
    ThreadInfo *info = (ThreadInfo *) argument;
    ...
    delete info;
    return NULL;
}

void run_threads(int N) {
    vector<pthread_t> threads(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void *) new ThreadInfo(...));
    }
    ...
}
```

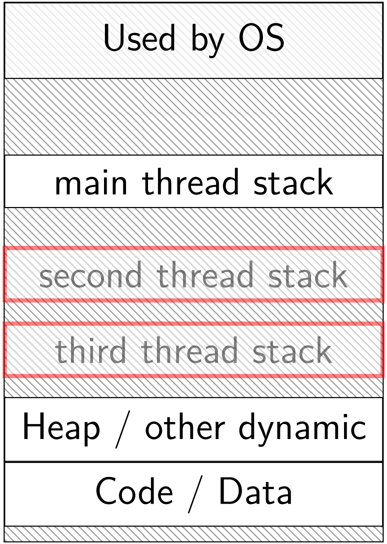
# what's wrong with this?

```
/* omitted: headers, using statements */
void *create_string(void *ignored_argument) {
  string result;
  result = ComputeString();
  return &result;
}
int main() {
  pthread_t the_thread;
  pthread_create(&the_thread, NULL, create_string, NULL);
  string *string_ptr;
  pthread_join(the_thread, &string_ptr);
  cout << "string is " << *string_ptr;
}
```

# program memory



# program memory




`0xFFFF FFFF FFFF FFFF`

`0xFFFF 8000 0000 0000`

`0x7F...`

dynamically allocated stacks
`string result` allocated here
`string_ptr` pointed to here

...stacks deallocated when threads exit/are joined

`0x0000 0000 0040 0000`

# thread resources

to create a thread, allocate:

new stack (how big???)

thread control block

pthreads: by default need to `join` thread to deallocate everything

thread kept around to allow collecting return value

# pthread_detach

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_create(&show_progress_thread, NULL, show_progress, NULL)
    pthread_detach(show_progress_thread);
}
int main() {
    spawn_show_progress_thread();
    do_other_stuff();
    ...
}
```

# starting threads detached

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setdetachstate(&attrs, PTHREAD_CREATE_DETACHED);
    pthread_create(&show_progress_thread, attrs,
                   show_progress, NULL);
    pthread_attr_destroy(&attrs);
}
```

# setting stack sizes

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setstacksize(&attrs, 32 * 1024 /* bytes */);
    pthread_create(&show_progress_thread, attrs,
                   show_progress, NULL);
}
```

# sum example (to global)

```
int values[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += values[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

# sum example (to global)

values, results: global variables — shared

```
int values[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += values[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

# sum example (to main stack, global values)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =  (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += values[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

# sum example (to main stack, global values)

```
int values[1024];
struct ThreadInfo
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =  (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += values[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

values: global variable — shared

# sum example (to main stack, global values)

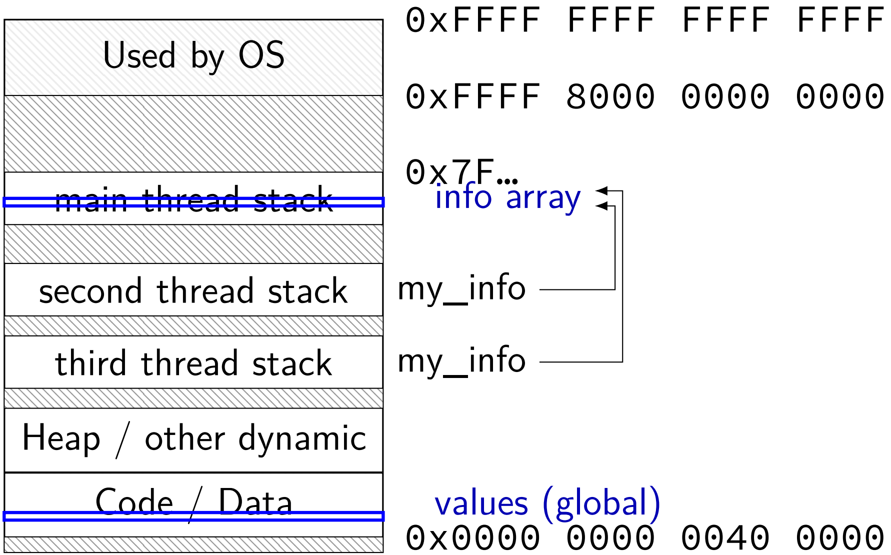
```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->
        sum += values[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

my_info: pointer to sum_all's stack
only okay because sum_all waits!

# sum example (to main stack, global values)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =  (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += values[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

# program memory (to main stack, global values)



# sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

# sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

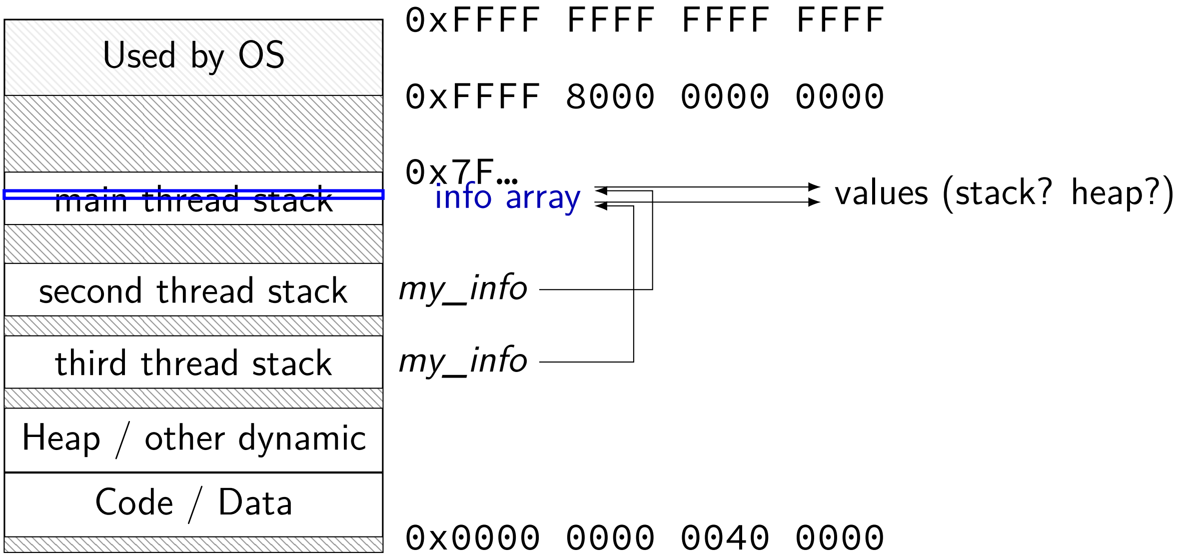
# sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

# sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

# program memory (to main stack)



# sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result }
void *sum_thread(void *argument) {
    ...
}
ThreadInfo *start_sum_all(int *values) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}
void finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

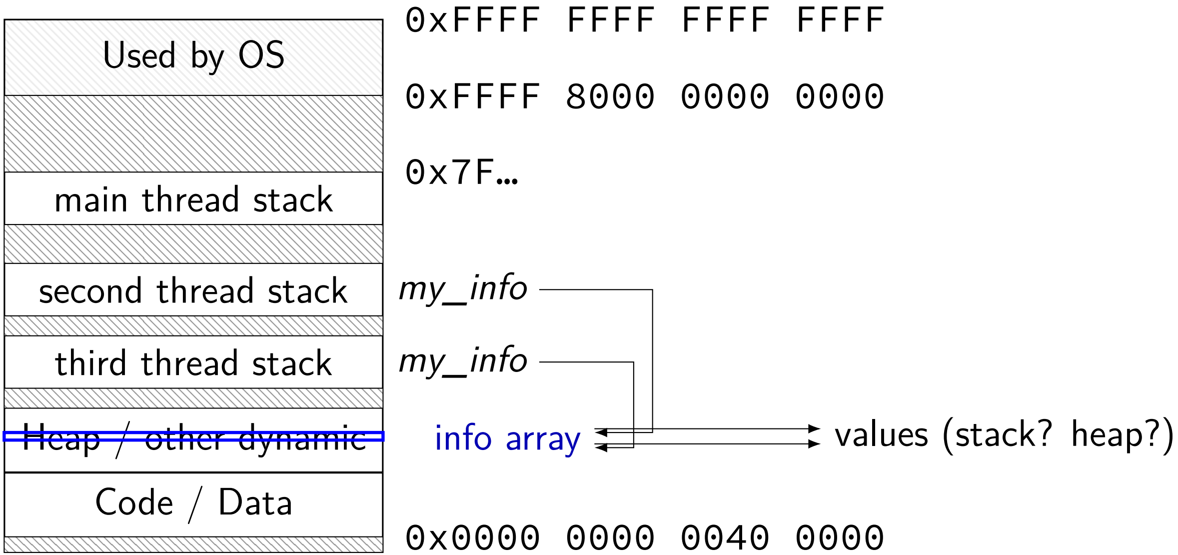
# sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result }
void *sum_thread(void *argument) {
    ...
}
ThreadInfo *start_sum_all(int *values) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}
void finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

# sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ...
}
ThreadInfo *start_sum_all(int *values) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}
void finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

# program memory (on heap)



# a note on error checking

from pthread_create manpage:

```
ERRORS
       EAGAIN Insufficient  resources  to  create  another thread, or a system-imposed limit on the number of
              threads was encountered.  The latter case may occur in two ways: the RLIMIT_NPROC soft resource
              limit  (set  via  setrlimit(2)),  which  limits  the  number of process for a real user ID, was
              reached; or the kernel's system-wide limit on the number of threads,  /proc/sys/kernel/threads-
              max, was reached.

       EINVAL Invalid settings in attr.

       EPERM  No permission to set the scheduling policy and parameters specified in attr.
```

special constants for *return value*

same pattern for many other pthreads functions

will often omit error checking in slides for brevity

# error checking pthread_create

```
int error = pthread_create(...);
if (error != 0) {
    /* print some error message */
}
```

# the correctness problem

schedulers introduce non-determinism

- scheduler might run threads in any order
- scheduler can switch threads at any time

worse with threads on multiple cores

- cores not precisely synchronized (stalling for caches, etc., etc.)
- different cores happen in different order each time

makes reliable testing very difficult

solution: correctness by design

# example application: ATM server

commands: withdraw, deposit

one correctness goal: don't lose money

# ATM server

(pseudocode)

```
ServerLoop() {
    while (true) {
        ReceiveRequest(&operation, &accountNumber, &amount);
        if (operation == DEPOSIT) {
            Deposit(accountNumber, amount);
        } else ...
    }
}
Deposit(accountNumber, amount) {
    account = GetAccount(accountId);
    account->balance += amount;
    StoreAccount(account);
}
```

# a threaded server?

```
Deposit(accountNumber, amount) {
    account = GetAccount(accountId);
    account->balance += amount;
    StoreAccount(account);
}
```

maybe Get/StoreAccount can be slow?

- read/write disk sometimes? contact another server sometimes?

maybe lots of requests to process?

- maybe real logic has more checks than Deposit()
- ...

all reasons to handle multiple requests at once

→ many threads all running the server loop

# multiple threads

```
main() {
    for (int i = 0; i < NumberOfThreads; ++i) {
        pthread_create(&server_loop_threads[i], NULL,
                       ServerLoop, NULL);
    }
    ...
}

ServerLoop() {
    while (true) {
        ReceiveRequest(&operation, &accountNumber, &amount);
        if (operation == DEPOSIT) {
            Deposit(accountNumber, amount);
        } else ...
    }
}
```

# a side note

why am I spending time justifying this?

multiple threads for something like this make things much trickier

we'll be learning why...

# the lost write

`account->balance += amount;` (in two threads, same account)

| Thread A | Thread B |
| --- | --- |
| `mov account->balance, %rax` | |
| `add amount, %rax` | |
| context switch | |
| | `mov account->balance, %rax` |
| | `add amount, %rax` |
| context switch | |
| `mov %rax, account->balance` | |
| context switch | |
| | `mov %rax, account->balance` |

# the lost write

`account->balance += amount;` (in two threads, same account)

| Thread A | | Thread B |
|---|---|---|
| `mov account->balance, %rax` | | |
| `add amount, %rax` | | |
| | context switch | |
| | | `mov account->balance, %rax` |
| | | `add amount, %rax` |
| | context switch | |
| `mov %rax, account->balance` | | |
| | context switch | |
| | | `mov %rax, account->balance` |

lost write to balance

"winner" of the race

# the lost write

`account->balance += amount;` (in two threads, same account)

| Thread A | | Thread B |
| --- | --- | --- |
| `mov account->balance, %rax`<br>`add amount, %rax` | | |
| | context switch | |
| | | `mov account->balance, %rax`<br>`add amount, %rax` |
| | context switch | |
| `mov %rax, account->balance` | | |
| | context switch | |
| | | `mov %rax, account->balance` |

lost write to balance

"winner" of the race

lost track of thread A's money

# thinking about race conditions (1)

what are the possible values of $x$?

(initially $x = y = 0$)

| **Thread A** | **Thread B** |
|---|---|
| $x \leftarrow 1$ | $y \leftarrow 2$ |

# thinking about race conditions (1)

what are the possible values of $x$?

(initially $x = y = 0$)

| **Thread A** | **Thread B** |
|---|---|
| $x \leftarrow 1$ | $y \leftarrow 2$ |

must be 1. Thread B can't do anything

# thinking about race conditions (2)

what are the possible values of $x$?

(initially $x = y = 0$)

| **Thread A** | **Thread B** |
|---|---|
| $x \leftarrow y + 1$ | $y \leftarrow 2$ |
| | $y \leftarrow y \times 2$ |

# thinking about race conditions (2)

what are the possible values of $x$?

(initially $x = y = 0$)

| **Thread A** | **Thread B** |
|---|---|
| $x \leftarrow y + 1$ | $y \leftarrow 2$ |
| | $y \leftarrow y \times 2$ |

1 or 3 or 5 (non-deterministic)

# thinking about race conditions (3)

what are the possible values of $x$?

(initially $x = y = 0$)

| **Thread A** | **Thread B** |
|---|---|
| $x \leftarrow 1$ | $x \leftarrow 2$ |

# thinking about race conditions (3)

what are the possible values of $x$?

(initially $x = y = 0$)

| Thread A | Thread B |
|---|---|
| $x \leftarrow 1$ | $x \leftarrow 2$ |

1 or 2

# thinking about race conditions (3)

what are the possible values of $x$?

(initially $x = y = 0$)

| **Thread A** | **Thread B** |
| --- | --- |
| $x \leftarrow 1$ | $x \leftarrow 2$ |

1 or 2

...but why not 3? maybe each bit of $x$ assigned seperately?

# atomic operation

*atomic operation* = operation that runs to completion or not at all

we will use these to let threads work together

most machines: loading/storing words is atomic

- so can't get $3$ from $x \leftarrow 1$ and $x \leftarrow 2$ running in parallel

but some instructions are not atomic

- one example: normal x86 add constant to memory

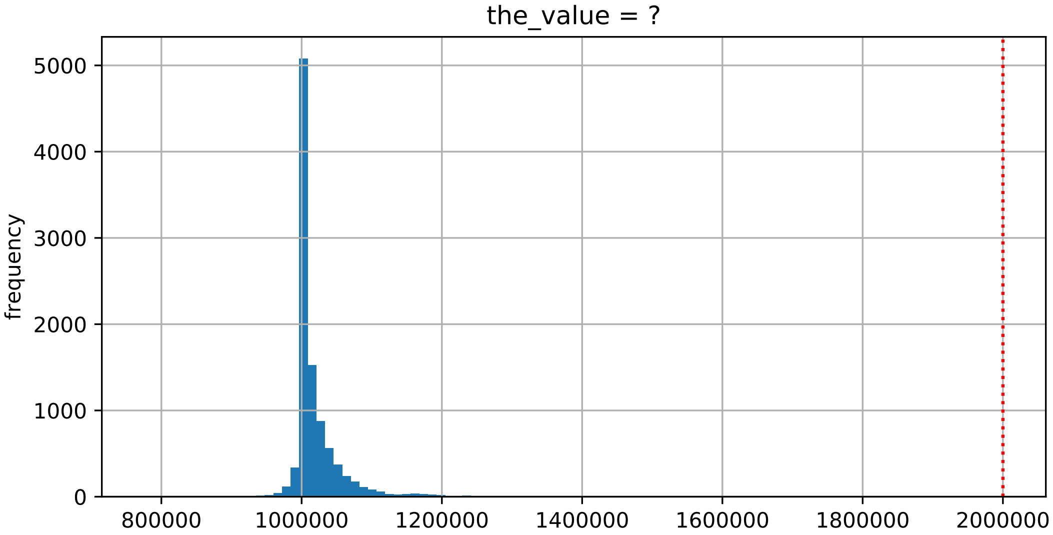
# lost adds (program)

```
.global update_loop
update_loop:
    addl $1, the_value // the_value (global variable) += 1
    dec %rdi           // argument 1 -= 1
    jg update_loop     // if argument 1 >= 0 repeat
    ret
```

---

```
int the_value;
extern void *update_loop(void *);
int main(void) {
    the_value = 0;
    pthread_t A, B;
    pthread_create(&A, NULL, update_loop, (void*) 1000000);
    pthread_create(&B, NULL, update_loop, (void*) 1000000);
    pthread_join(A, NULL);
    pthread_join(B, NULL);
    // expected result: 1000000 + 1000000 = 2000000
    printf("the_value = %d\n", the_value);
}
```

# lost adds (results)



# but how?

probably not possible on single core

- exceptions can't occur in the middle of `add` instruction

…but 'add to memory' implemented with multiple steps

- still needs to load, add, store internally
- can be interleaved with what other cores do

# but how?

probably not possible on single core

- exceptions can't occur in the middle of `add` instruction

…but 'add to memory' implemented with multiple steps

- still needs to load, add, store internally
- can be interleaved with what other cores do

(and actually it's more complicated than that — we'll talk later)

# so, what is actually atomic

for now we'll assume: load/stores of 'words'

(64-bit machine = 64-bits words)

in general: processor designer will tell you

their job to design caches, etc. to work as documented

# too much milk

roommates Alice and Bob want to keep fridge stocked with milk:

| time | Alice | Bob |
|---|---|---|
| 3:00 | look in fridge. no milk | |
| 3:05 | leave for store | |
| 3:10 | arrive at store | look in fridge. no milk |
| 3:15 | buy milk | leave for store |
| 3:20 | return home, put milk in fridge | arrive at store |
| 3:25 | | buy milk |
| 3:30 | | return home, put milk in fridge |

how can Alice and Bob coordinate better?

# too much milk “solution” 1 (algorithm)

leave a note: “I am buying milk”

- place before buying
- remove after buying
- don’t try buying if there’s a note

≈ setting/checking a variable (e.g. “`note = 1`”)

- with atomic load/store of variable

```
if (no milk) {
    if (no note) {
        leave note;
        buy milk;
        remove note;
    }
}
```

# too much milk "solution" 1 (timeline)

**Alice**

```
if (no milk) {
    if (no note) {



        leave note;
        buy milk;
        remove note;
    }
}
```

**Bob**

```



if (no milk) {
    if (no note) {





        leave note;
        buy milk;
        remove note;
    }
}
```

# too much milk “solution” 2 (algorithm)

intuition: leave note when buying or checking if need to buy

```
leave note;
if (no milk) {
    if (no note) {
        buy milk;
    }
}
remove note;
```

# too much milk: “solution” 2 (timeline)

**Alice**

```
leave note;
if (no milk) {
    if (no note) {
        buy milk;
    }
}
remove note;
```

# too much milk: “solution” 2 (timeline)

**Alice**

```
leave note;
if (no milk) {
    if (no note) {
        buy milk;
    }
}
remove note;
```

← but there’s always a note

# too much milk: “solution” 2 (timeline)

**Alice**

```
leave note;
if (no milk) {
    if (no note) {
        buy milk;
    }
}
remove note;
```

but there’s always a note

…will never buy milk (twice *or* once)

# “solution” 3: algorithm

intuition: label notes so Alice knows which is hers (and vice-versa)

computer equivalent: separate noteFromAlice and noteFromBob variables

**Alice**

```
leave note from Alice;
if (no milk) {
    if (no note from Bob) {
        buy milk
    }
}
remove note from Alice;
```

**Bob**

```
leave note from Bob;
if (no milk) {
    if (no note from Alice)
        buy milk
    }
}
remove note from Bob;
```

# too much milk: “solution” 3 (timeline)

**Alice**

```
leave note from Alice
if (no milk) {
    if (no note from Bob) {
        ~~buy milk~~
    }
}
remove note from Alice
```

**Bob**

```
leave note from Bob
if (no milk) {
    if (no note from Alice) {
        ~~buy milk~~
    }
}
remove note from Bob
```

# too much milk: is it possible

is there a solutions with writing/reading notes?

$\approx$ loading/storing from shared memory

yes, but it's not very elegant

# too much milk: solution 4 (algorithm)

**Alice**

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

**Bob**

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

# too much milk: solution 4 (algorithm)

**Alice**

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

**Bob**

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

exercise (hard): prove (in)correctness

# too much milk: solution 4 (algorithm)

**Alice**

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

**Bob**

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

exercise (hard): prove (in)correctness

# too much milk: solution 4 (algorithm)

**Alice**

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

**Bob**

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

exercise (hard): prove (in)correctness

exercise (hard): extend to three people

# Peterson's algorithm

general version of solution

see, e.g., Wikipedia

we'll use special hardware support instead

# some definitions

**mutual exclusion**: ensuring only one thread does a particular thing at a time

like checking for and, if needed, buying milk

# some definitions

**mutual exclusion**: ensuring only one thread does a particular thing at a time

- like checking for and, if needed, buying milk

**critical section**: code that exactly one thread can execute at a time

- result of critical section

# some definitions

**mutual exclusion**: ensuring only one thread does a particular thing at a time

- like checking for and, if needed, buying milk

**critical section**: code that exactly one thread can execute at a time

- result of critical section

**lock**: object only one thread can hold at a time

- interface for creating critical sections

# the lock primitive

locks: an object with (at least) two operations:

*acquire* or *lock* — wait until lock is free, then “grab” it
*release* or *unlock* — let others use lock, wakeup waiters

typical usage: everyone acquires lock before using shared resource

forget to acquire lock? weird things happen

```
Lock(MilkLock);
if (no milk) {
    buy milk
}
Unlock(MilkLock);
```

# pthread mutex

```
#include <pthread.h>

pthread_mutex_t MilkLock;
pthread_mutex_init(&MilkLock, NULL);
...
pthread_mutex_lock(&MilkLock);
if (no milk) {
    buy milk
}
pthread_mutex_unlock(&MilkLock);
```

# xv6 spinlocks

```
#include "spinlock.h"
...
struct spinlock MilkLock;
initlock(&MilkLock, "name for debugging");
...
acquire(&MilkLock);
if (no milk) {
    buy milk
}
release(&MilkLock);
```

# C++ containers and locking

can you use a vector from multiple threads?

...question: how is it implemented?

# C++ containers and locking

can you use a vector from multiple threads?

...question: how is it implemented?

- dynamically allocated array
- reallocated on size changes

# C++ containers and locking

can you use a vector from multiple threads?

…question: how is it implemented?

- dynamically allocated array
- reallocated on size changes

can access from multiple threads …as long as not being resized?

# C++ standard rules for containers

multiple threads can read anything at the same time

can only read element if no other thread is modifying it

can only add/remove elements if no other threads are accessing container

some exceptions, read documentation really carefully

# implementing locks: single core

intuition: context switch only happens on interrupt

- timer expiration, I/O, etc. causes OS to run

solution: disable them

- reenable on unlock

# implementing locks: single core

intuition: context switch only happens on interrupt

- timer expiration, I/O, etc. causes OS to run

solution: disable them

- reenable on unlock

x86 instructions:

- `cli` — disable interrupts
- `sti` — enable interrupts

# naive interrupt enable/disable (1)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

# naive interrupt enable/disable (1)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

problem: user can hang the system:

```
Lock(some_lock);
while (true) {}
```

# naive interrupt enable/disable (1)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

problem: user can hang the system:

```
Lock(some_lock);
while (true) {}
```

problem: can't do I/O within lock

```
Lock(some_lock);
read from disk
    /* waits forever for (disabled) interrupt
       from disk IO finishing */
```

# naive interrupt enable/disable (2)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

# naive interrupt enable/disable (2)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

# naive interrupt enable/disable (2)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

# naive interrupt enable/disable (2)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

problem: nested locks

```
Lock(milk_lock);
if (no milk) {
    Lock(store_lock);
    buy milk
    Unlock(store_lock);
    /* interrupts enabled here?? */
}
Unlock(milk_lock);
```

# xv6 interrupt disabling (1)

```
...
acquire(struct spinlock *lk) {
  pushcli(); // disable interrupts to avoid deadlock
  ... /* this part basically just for multicore */
}
release(struct spinlock *lk)
{
  ... /* this part basically just for multicore */
  popcli();
}
```

# xv6 push/popcli

pushcli / popcli — need to be in pairs

pushcli — disable interrupts if not already

popcli — enable interrupts if corresponding pushcli disabled them

- don't enable them if they were already disabled

# backup slides

# thread versus process state

thread state — kept in **thread control block**

- registers (including program counter)
- other information?

process state — kept in **process control block**

- address space (memory layout)
- open files
- process id
- ...

# Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared — if same process

`fork()`-like system call "clone": choose what to share

- `clone(CLONE_FILES, ...)` — new process **sharing** open files
- `clone(CLONE_VM, ...)` — new process **sharing** address spaces

# Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared — if same process

`fork()`-like system call "clone": choose what to share

- `clone(CLONE_FILES, ...)` — new process **sharing** open files
- `clone(CLONE_VM, ...)` — new process **sharing** address spaces

advantage: no special logic for threads (mostly)

# aside: alternate threading models

we'll talk about kernel threads

OS scheduler deals directly with threads

alternate idea: library code handles threading

kernel doesn't know about threads w/in process

hierarchy of schedulers: one for processes, one within each process

not currently common model — awkward with multicore

# why threads?

concurrency: different things happening at once

- one thread per user of web server?
- one thread per page in web browser?
- one thread to play audio, one to read keyboard, ...?
- ...

parallelism: do same thing with more resources

- multiple processors to speed-up simulation (life assignment)